cs280s21-b.sg

Submitted on Friday, January 29, 2021, 7:39 PM (Download)

Automatic evaluation[-]

Proposed grade: 100 / 100

Compilation[-]
Running your code.....
Executable successfully created

Comments[-]
[+]Summary of tests

ObjectAllocator.h

```cpp
//---------------------------------------------------------------------------
#ifndef OBJECTALLOCATORH
#define OBJECTALLOCATORH
//---------------------------------------------------------------------------

#include <string>

// If the client doesn't specify these:
static const int DEFAULT_OBJECTS_PER_PAGE = 4;
static const int DEFAULT_MAX_PAGES = 3;

/*!
  Exception class
*/
class OAException
{
  public:
    /*!
      Possible exception codes
    */
    enum OA_EXCEPTION
    {
      E_NO_MEMORY,      //!< out of physical memory (operator new fails)
      E_NO_PAGES,       //!< out of logical memory (max pages has been reached)
      E_BAD_BOUNDARY,   //!< block address is on a page, but not on any block-boundary
      E_MULTIPLE_FREE,  //!< block has already been freed
      E_CORRUPTED_BLOCK //!< block has been corrupted (pad bytes have been overwritten)
    };

    /*!
      Constructor

      \param ErrCode
        One of the 5 error codes listed above

      \param Message
        A message returned by the what method.
    */
    OAException(OA_EXCEPTION ErrCode, const std::string& Message) : error_code_(ErrCode), message_(Message) {};

    /*!
      Destructor
    */
    virtual ~OAException() {
    }

    /*!
      Retrieves the error code

      \return
        One of the 5 error codes.
    */
    OA_EXCEPTION code() const {
      return error_code_;
    }

    /*!
      Retrieves a human-readable string regarding the error.

      \return
        The NUL-terminated string representing the error.
    */
    virtual const char *what() const {
      return message_.c_str();
    }
  private:
    OA_EXCEPTION error_code_; //!< The error code (one of the 5)
    std::string message_;     //!< The formatted string for the user.
};


/*!
  ObjectAllocator configuration parameters
*/
struct OAConfig
{
  static const size_t BASIC_HEADER_SIZE = sizeof(unsigned) + 1; //!< allocation number + flags
  static const size_t EXTERNAL_HEADER_SIZE = sizeof(void*);     //!< just a pointer

  /*!
    The different types of header blocks
  */
  enum HBLOCK_TYPE{hbNone, hbBasic, hbExtended, hbExternal};

  /*!
    POD that stores the information related to the header blocks.
  */
  struct HeaderBlockInfo
  {
    HBLOCK_TYPE type_;  //!< Which of the 4 header types to use?
    size_t size_;       //!< The size of this header
    size_t additional_; //!< How many user-defined additional bytes

    /*!
      Constructor

      \param type
        The kind of header blocks in use.

      \param additional
        The number of user-defined additional bytes required.

    */
    HeaderBlockInfo(HBLOCK_TYPE type = hbNone, unsigned additional = 0) : type_(type), size_(0), additional_(additional)
    {
      if (type_ == hbBasic)
        size_ = BASIC_HEADER_SIZE;
```

```
      else if (type_ == hbExtended) // alloc # + use counter + flag byte + user-defined
        size_ = sizeof(unsigned int) + sizeof(unsigned short) + sizeof(char) + additional_;
      else if (type_ == hbExternal)
        size_ = EXTERNAL_HEADER_SIZE;
    };
  };

  /*!
    Constructor

    \param UseCPPMemManager
      Determines whether or not to by-pass the OA.

    \param ObjectsPerPage
      Number of objects for each page of memory.

    \param MaxPages
      Maximum number of pages before throwing an exception. A value
      of 0 means unlimited.

    \param DebugOn
      Is debugging code on or off?

    \param PadBytes
      The number of bytes to the left and right of a block to pad with.

    \param HBInfo
      Information about the header blocks used

    \param Alignment
      The number of bytes to align on.
  */
  OAConfig(bool UseCPPMemManager = false,
           unsigned ObjectsPerPage = DEFAULT_OBJECTS_PER_PAGE,
           unsigned MaxPages = DEFAULT_MAX_PAGES,
           bool DebugOn = false,
           unsigned PadBytes = 0,
           const HeaderBlockInfo &HBInfo = HeaderBlockInfo(),
           unsigned Alignment = 0) : UseCPPMemManager_(UseCPPMemManager),
                                     ObjectsPerPage_(ObjectsPerPage),
                                     MaxPages_(MaxPages),
                                     DebugOn_(DebugOn),
                                     PadBytes_(PadBytes),
                                     HBlockInfo_(HBInfo),
                                     Alignment_(Alignment)
  {
    HBlockInfo_ = HBInfo;
    LeftAlignSize_ = 0;
    InterAlignSize_ = 0;
  }

  bool UseCPPMemManager_;      //!< by-pass the functionality of the OA and use new/delete
  unsigned ObjectsPerPage_;    //!< number of objects on each page
  unsigned MaxPages_;          //!< maximum number of pages the OA can allocate (0=unlimited)
  bool DebugOn_;               //!< enable/disable debugging code (signatures, checks, etc.)
  unsigned PadBytes_;          //!< size of the left/right padding for each block
  HeaderBlockInfo HBlockInfo_; //!< size of the header for each block (0=no headers)
  unsigned Alignment_;         //!< address alignment of each block
  unsigned LeftAlignSize_;     //!< number of alignment bytes required to align first block
  unsigned InterAlignSize_;    //!< number of alignment bytes required between remaining blocks
};


/*!
  POD that holds the ObjectAllocator statistical info
*/
struct OAStats
{
  /*!
    Constructor
  */
  OAStats() : ObjectSize_(0), PageSize_(0), FreeObjects_(0), ObjectsInUse_(0), PagesInUse_(0),
              MostObjects_(0), Allocations_(0), Deallocations_(0) {};

  size_t ObjectSize_;      //!< size of each object
  size_t PageSize_;        //!< size of a page including all headers, padding, etc.
  unsigned FreeObjects_;   //!< number of objects on the free list
  unsigned ObjectsInUse_;  //!< number of objects in use by client
  unsigned PagesInUse_;    //!< number of pages allocated
  unsigned MostObjects_;   //!< most objects in use by client at one time
  unsigned Allocations_;   //!< total requests to allocate memory
  unsigned Deallocations_; //!< total requests to free memory
};

/*!
  This allows us to easily treat raw objects as nodes in a linked list
*/
struct GenericObject
{
  GenericObject *Next; //!< The next object in the list
};

/*!
  This is used with external headers
*/
struct MemBlockInfo
{
  bool in_use;        //!< Is the block free or in use?
  char *label;        //!< A dynamically allocated NUL-terminated string
  unsigned alloc_num; //!< The allocation number (count) of this block
};

/*!
  This class represents a custom memory manager
*/
class ObjectAllocator
{
```

```
215  public:
216    // Defined by the client (pointer to a block, size of block)
217    typedef void (*DUMPCALLBACK)(const void *, size_t);     //!< Callback function when dumping memory leaks
218    typedef void (*VALIDATECALLBACK)(const void *, size_t); //!< Callback function when validating blocks
219
220    // Predefined values for memory signatures
221    static const unsigned char UNALLOCATED_PATTERN = 0xAA; //!< New memory never given to the client
222    static const unsigned char ALLOCATED_PATTERN =   0xBB; //!< Memory owned by the client
223    static const unsigned char FREED_PATTERN =       0xCC; //!< Memory returned by the client
224    static const unsigned char PAD_PATTERN =         0xDD; //!< Pad signature to detect buffer over/under flow
225    static const unsigned char ALIGN_PATTERN =       0xEE; //!< For the alignment bytes
226
227    // Creates the ObjectManager per the specified values
228    // Throws an exception if the construction fails. (Memory allocation problem)
229    ObjectAllocator(size_t ObjectSize, const OAConfig& config);
230
231    // Destroys the ObjectManager (never throws)
232    ~ObjectAllocator();
233
234    // Take an object from the free list and give it to the client (simulates new)
235    // Throws an exception if the object can't be allocated. (Memory allocation problem)
236    void *Allocate(const char *label = 0);
237
238    // Returns an object to the free list for the client (simulates delete)
239    // Throws an exception if the the object can't be freed. (Invalid object)
240    void Free(void *Object);
241
242    // Calls the callback fn for each block still in use
243    unsigned DumpMemoryInUse(DUMPCALLBACK fn) const;
244
245    // Calls the callback fn for each block that is potentially corrupted
246    unsigned ValidatePages(VALIDATECALLBACK fn) const;
247
248    // Frees all empty page
249    unsigned FreeEmptyPages();
250
251    // Testing/Debugging/Statistic methods
252    void SetDebugState(bool State);   // true=enable, false=disable
253    const void *GetFreeList() const;  // returns a pointer to the internal free list
254    const void *GetPageList() const;  // returns a pointer to the internal page list
255    OAConfig GetConfig() const;       // returns the configuration parameters
256    OAStats GetStats() const;         // returns the statistics for the allocator
257
258    // Prevent copy construction and assignment
259    ObjectAllocator(const ObjectAllocator &oa) = delete;            //!< Do not implement!
260    ObjectAllocator &operator=(const ObjectAllocator &oa) = delete; //!< Do not implement!
261
262  private:
263
264    OAConfig OAConfig_;
265    OAStats OAStats_;
266
267    // Some "suggested" members (only a suggestion!)
268    GenericObject *PageList_; //!< the beginning of the list of pages
269    GenericObject *FreeList_; //!< the beginning of the list of objects
270
271    size_t midBlockSize;
272    size_t offset;
273
274    //add page to the list
275    void addPage();
276    void updateOAStats();
277
278    size_t getPageSize();
279    bool isPageEmpty(GenericObject* page) const;
280    void freePage(GenericObject* temp);
281    void removePgObjInFreeList(GenericObject* pageAddress);
282    bool inPage(GenericObject* pageAddress, unsigned char* address) const;
283
284  };
285
```

ObjectAllocator.cpp

```cpp
/*****************************************************************************/
/*!
\file:      ObjectAllocator.cpp
\author:    Goh Wei Zhe, weizhe.goh, 440000119
\par email: weizhe.goh\@digipen.edu
\date:      January 21, 2021
\brief      To learn about and understand data structures and their
            interfaces.

Copyright (C) 2021 DigiPen Institute of Technology.
Reproduction or disclosure of this file or its contents without the
prior written consent of DigiPen Institute of Technology is prohibited.
*/
/*****************************************************************************/

#include "ObjectAllocator.h"
#include <cstring>

#define u_char unsigned char
/*****************************************************************************/
/*!
\fn     ObjectAllocator::ObjectAllocator(size_t ObjectSize,
        const OAConfig& config)

\brief  Constructor to calculate and initialise class variables

\param  ObjectSize - the size of the object allocateed
\param  config - contains the OAConfig details for object allocator
*/
/*****************************************************************************/
ObjectAllocator::ObjectAllocator(size_t ObjectSize, const OAConfig& config)
    :OAConfig_{ config }, OAStats_{}, PageList_{ nullptr }, FreeList_{nullptr},
    midBlockSize{0}, offset{0}
{
    OAStats_.ObjectSize_ = ObjectSize;
    OAStats_.PageSize_ = getPageSize();

    //calculate size of block
    midBlockSize = OAConfig_.HBlockInfo_.size_ + OAConfig_.PadBytes_ * 2 +
    OAStats_.ObjectSize_ + OAConfig_.InterAlignSize_;

    //calculate page size
    OAStats_.PageSize_ = sizeof(void*) + OAConfig_.LeftAlignSize_ +
    (OAConfig_.ObjectsPerPage_ * midBlockSize) - OAConfig_.InterAlignSize_;

    //calculate offset to start at s
    //offset = P + al + h + s
    offset = sizeof(void*) + OAConfig_.LeftAlignSize_ +
        OAConfig_.HBlockInfo_.size_ + OAConfig_.PadBytes_;

     //if memory is allocated
    if(OAConfig_.UseCPPMemManager_)
        return;

    //Page Allocation
    addPage();
}

/*****************************************************************************/
/*!
\fn     ObjectAllocator::~ObjectAllocator()
\brief  Destructor for ObjectAllocator. Remove pages.
*/
/*****************************************************************************/
ObjectAllocator::~ObjectAllocator()
{
    u_char* page;

    while (PageList_ != NULL)
    {
        page = reinterpret_cast<u_char*>(PageList_);

        PageList_ = PageList_->Next;
        delete [] page;
    }
}

/*****************************************************************************/
/*!
\fn     ObjectAllocator::Allocate(const char* label)

\brief  Allocates memory for page and return to client. If UseCPPMemManager is
         true, bypass OA and use new.

\param  label - the size of the object allocateed

\return A pointer to a block of object
*/
/*****************************************************************************/
void* ObjectAllocator::Allocate(const char* label)
{
    if(OAConfig_.UseCPPMemManager_)
    {
        try
        {
            u_char* newObj = new u_char[OAStats_.ObjectSize_];
            updateOAStats();

            return newObj;
        }
        catch (std::bad_alloc& )
        {
            throw OAException(OAException::E_NO_MEMORY, "Not enough memory!");
        }
    }

    //Freelist is empty, no page allocated
```

```
108     if(FreeList_ == nullptr)
109     {
110         if(OAConfig_.MaxPages_ == 0 || OAStats_.PagesInUse_<OAConfig_.MaxPages_)
111         {
112             addPage();
113         }
114         else
115         {
116             throw OAException(OAException::E_NO_PAGES, "No available page!");
117         }
118     }
119 
120     GenericObject* object = FreeList_;
121     FreeList_ = FreeList_->Next;
122 
123     if(OAConfig_.DebugOn_)
124     {
125         memset(object, ALLOCATED_PATTERN, OAStats_.ObjectSize_);
126     }
127 
128     updateOAStats();
129 
130    switch(OAConfig_.HBlockInfo_.type_)
131    {
132        case OAConfig::hbBasic:
133        {
134            //Start of header
135            u_char* headerStart = reinterpret_cast<u_char*>(object) -
136             OAConfig_.PadBytes_ - OAConfig_.HBlockInfo_.size_;
137 
138            unsigned* allocationNumber =reinterpret_cast<unsigned*>(headerStart);
139            *allocationNumber = OAStats_.Allocations_;
140 
141            //Start of flag
142            u_char* flag = reinterpret_cast<u_char*>(headerStart +
143             sizeof(unsigned));
144 
145            *flag = true;
146 
147            break;
148        }
149        case OAConfig::hbExtended:
150        {
151            //Start of Flag
152            u_char* flagStart = reinterpret_cast<u_char*>(object) -
153            OAConfig_.PadBytes_ - sizeof(u_char);
154 
155            *flagStart = true;
156 
157            //Start of allocation number
158            u_char* allocNumStart = flagStart - sizeof(unsigned);
159 
160            unsigned *allocationNumber =
161            reinterpret_cast<unsigned*>(allocNumStart);
162 
163            *allocationNumber = OAStats_.Allocations_;
164 
165            //Start of use-count
166            u_char* useCountStart = allocNumStart - sizeof(unsigned short);
167 
168            unsigned short* useCount =
169            reinterpret_cast<unsigned short*>(useCountStart);
170 
171            ++(*useCount);
172 
173            break;
174        }
175        case OAConfig::hbExternal:
176        {
177            //one unsigned for allocation number, one flag to determine on off
178            u_char* headerStart = reinterpret_cast<u_char*>(object) -
179            OAConfig_.PadBytes_ - OAConfig_.HBlockInfo_.size_;
180 
181            MemBlockInfo** memPtr =
182            reinterpret_cast <MemBlockInfo**>(headerStart);
183 
184            try
185            {
186                *memPtr = new MemBlockInfo;
187 
188                (*memPtr)->in_use = true;
189                (*memPtr)->label = nullptr;
190                (*memPtr)->alloc_num = OAStats_.Allocations_;
191 
192                if(label)
193                {
194                    try
195                    {
196                        (*memPtr)->label = new char[strlen(label) + 1];
197                    }
198                    catch (std::bad_alloc& )
199                    {
200                        throw OAException(OAException::E_NO_MEMORY,
201                        "Out of memory!");
202                    }
203 
204                    strcpy((*memPtr)->label, label);
205                }
206            }
207            catch(std::bad_alloc& )
208            {
209                throw OAException(OAException::E_NO_MEMORY, "Out of memory!");
210            }
211            break;
212        }
213 
214        default: break;
```

```
215     }
216
217     return object;
218 }
219
220 /*************************************************************************/
221 /*!
222 \fn     void ObjectAllocator::Free(void* Object)
223
224 \brief  Free block memory from Allocate() If UseCPPMemManager is true, bypass
225         OA and use delete. If debug is on, perform check to see if block is
226         corrupted and within same boundaries.
227
228 \param  Object - A pointer to object allocated from Allocate().
229 */
230 /*************************************************************************/
231 void ObjectAllocator::Free(void* Object)
232 {
233     if(OAConfig_.UseCPPMemManager_)
234     {
235         delete[] reinterpret_cast<u_char*>(Object);
236
237         --OAStats_.ObjectsInUse_;
238         ++OAStats_.Deallocations_;
239         ++OAStats_.FreeObjects_;
240
241         return;
242     }
243
244     GenericObject* page = PageList_;
245     u_char* objAddress = reinterpret_cast<u_char*>(Object);
246
247     //Only perform check when debugging
248     if(OAConfig_.DebugOn_)
249     {
250         bool onBound = false;
251         bool corrupted = false;
252         bool alreadyFreed = true;
253
254         while(page!= NULL)
255         {
256             if( reinterpret_cast<u_char*>(page) < objAddress &&
257             objAddress < reinterpret_cast<u_char*>(page) + OAStats_.PageSize_)
258             {
259                 u_char* blockStart = reinterpret_cast<u_char*>(page) + offset;
260
261                 size_t separation = midBlockSize;
262
263                 if((objAddress - blockStart) % separation == 0)
264                 {
265                     onBound = true;
266                 }
267             }
268
269             page = page->Next;
270         }
271
272         if(onBound == false)
273         {
274             throw(OAException(OAException::E_BAD_BOUNDARY,
275             "Object not on a block boundary"));
276         }
277         else
278         {
279             //check data block
280             for(unsigned i = sizeof(u_char*); i < OAStats_.ObjectSize_; ++i)
281             {
282                 if(objAddress[i] != FREED_PATTERN)
283                 {
284                     alreadyFreed = false;
285                     break;
286                 }
287             }
288
289             //Check padding blocks
290             u_char* leftPad = objAddress - OAConfig_.PadBytes_;
291             u_char* rightPad = objAddress + OAStats_.ObjectSize_;
292
293             for(unsigned int i = 0; i < OAConfig_.PadBytes_; ++i)
294             {
295                 if(leftPad[i] != PAD_PATTERN || rightPad[i] != PAD_PATTERN)
296                 {
297                     corrupted = true;
298                     break;
299                 }
300             }
301         }
302
303         if(alreadyFreed == true)
304         {
305             throw(OAException(OAException::E_MULTIPLE_FREE,
306             "Block already freed once"));
307         }
308         else if (corrupted == true)
309         {
310             throw(OAException(OAException::E_CORRUPTED_BLOCK,
311             "Block is corrupted"));
312         }
313     }
314
315     u_char* headerStart = reinterpret_cast<u_char*>(objAddress) -
316     OAConfig_.PadBytes_ - OAConfig_.HBlockInfo_.size_;
317
318     switch(OAConfig_.HBlockInfo_.type_)
319     {
320         case OAConfig::hbBasic:
321         {
```

```
            memset(headerStart, 0, OAConfig::BASIC_HEADER_SIZE);
            break;
        }
        case OAConfig::hbExtended:
        {
            memset(headerStart + OAConfig_.HBlockInfo_.additional_ +
            sizeof(unsigned short), 0, OAConfig::BASIC_HEADER_SIZE);

            break;
        }
        case OAConfig::hbExternal:
        {
            MemBlockInfo** memPtr =
            reinterpret_cast<MemBlockInfo**>(headerStart);

            delete[] (*memPtr)->label;
            delete *memPtr;

            memset(memPtr, 0, OAConfig_.HBlockInfo_.size_);
        }
        default: break;
    }

    if(OAConfig_.DebugOn_)
    {
        memset(Object, FREED_PATTERN, OAStats_.ObjectSize_);
    }

    GenericObject* object = reinterpret_cast<GenericObject*>(Object);
    object->Next = FreeList_;
    FreeList_ = object;

    --OAStats_.ObjectsInUse_;
    ++OAStats_.Deallocations_;
    ++OAStats_.FreeObjects_;
}

/*************************************************************************/
/*!
\fn     unsigned ObjectAllocator::DumpMemoryInUse(DUMPCALLBACK fn) const

\brief  Run through pages in OA and calls the given function if the block
        data are active.

\param  fn - A function to call for each active object.

\return Number of active objects
*/
/*************************************************************************/
unsigned ObjectAllocator::DumpMemoryInUse(DUMPCALLBACK fn) const
{
    GenericObject* page = PageList_;

    u_char* block;

    while (page)
    {
        block = reinterpret_cast<u_char*>(page) + offset;

        for (unsigned i = 0; i < OAConfig_.ObjectsPerPage_; ++i)
        {
          if (*block == ALLOCATED_PATTERN)
          {
            fn(block, OAStats_.ObjectSize_);
          }
          if (i < OAConfig_.ObjectsPerPage_ - 1)
          {
            block += midBlockSize;
          }
        }

        page = page->Next;
    }

    return OAStats_.ObjectsInUse_;
}

/*************************************************************************/
/*!
\fn     unsigned ObjectAllocator::ValidatePages(VALIDATECALLBACK fn) const

\brief  Run through each pages to check if each data block is corrupted or not.

\param  fn - A function to call once a data block found corrupted.

\return Number of corrupted data blocks.
*/
/*************************************************************************/
unsigned ObjectAllocator::ValidatePages(VALIDATECALLBACK fn) const
{
  if (OAConfig_.PadBytes_ == 0)
  {
    return 0;
  }

  unsigned corruptedBlocks = 0;
  GenericObject* page = PageList_;

  u_char* block;
  u_char* leftPad;
  u_char* rightPad;

  while (page)
  {
    block = reinterpret_cast<u_char*>(page) + offset;

    for (unsigned i = 0; i < OAConfig_.ObjectsPerPage_; ++i)
```

```
    {
      // Check padding blocks
      leftPad = block - OAConfig_.PadBytes_;
      rightPad = block + OAStats_.ObjectSize_;

      for (unsigned j = 0; j < OAConfig_.PadBytes_; ++j)
      {
        if (leftPad[j] != PAD_PATTERN || rightPad[j] != PAD_PATTERN)
        {
          fn(block, OAStats_.ObjectSize_);
          ++corruptedBlocks;
          break;
        }
      }

      // Prevents pointer arithmetic overrun
      if (i < OAConfig_.ObjectsPerPage_ - 1)
      {
        block += midBlockSize;
      }
    }
    page = page->Next;
  }

  return corruptedBlocks;
}

/*****************************************************************************/
/*!
\fn     unsigned ObjectAllocator::FreeEmptyPages()

\brief  Run through each page to check if page is empty. If empty, returns
        memory back to OS.

\return Number of page freed.
*/
/*****************************************************************************/
unsigned ObjectAllocator::FreeEmptyPages()
{
    if(PageList_ == nullptr)
        return 0;

    unsigned emptyPages = 0;

    GenericObject* temp = PageList_;
    GenericObject* previous = nullptr;

    //If head node holds a key or multiple key
    while(temp != nullptr && isPageEmpty(temp))
    {
        //Change head
        PageList_ = temp->Next;
        freePage(temp);

        //Change temp
        temp = PageList_;
        ++emptyPages;
    }

    //Delete occurance other than head
    while(temp != nullptr)
    {
      //Search key to be deleted
      //Keep track of previous node as we need to change previous->Next
      while(temp != nullptr && !isPageEmpty(temp))
      {
        previous = temp;
        temp = temp->Next;
      }

      //If key not present in linked list
      if(temp == nullptr)
        return emptyPages;

      //Unlink node from linked list
      previous->Next = temp->Next;
      freePage(temp);

      //Update temp for next loop
      temp = previous->Next;
      ++emptyPages;
    }
    return emptyPages;
}

/*****************************************************************************/
/*!
\fn     void ObjectAllocator::SetDebugState(bool State)

\brief  Set current OA to the given debug state.

\param  State - Debug on (True), Debug off (False)

*/
/*****************************************************************************/
void ObjectAllocator::SetDebugState(bool State)
{
    OAConfig_.DebugOn_ = State;
}

/*****************************************************************************/
/*!
\fn     const void* ObjectAllocator::GetFreeList() const

\brief  Returns the free list, a linked list to determine which block of memory
        are free.
```

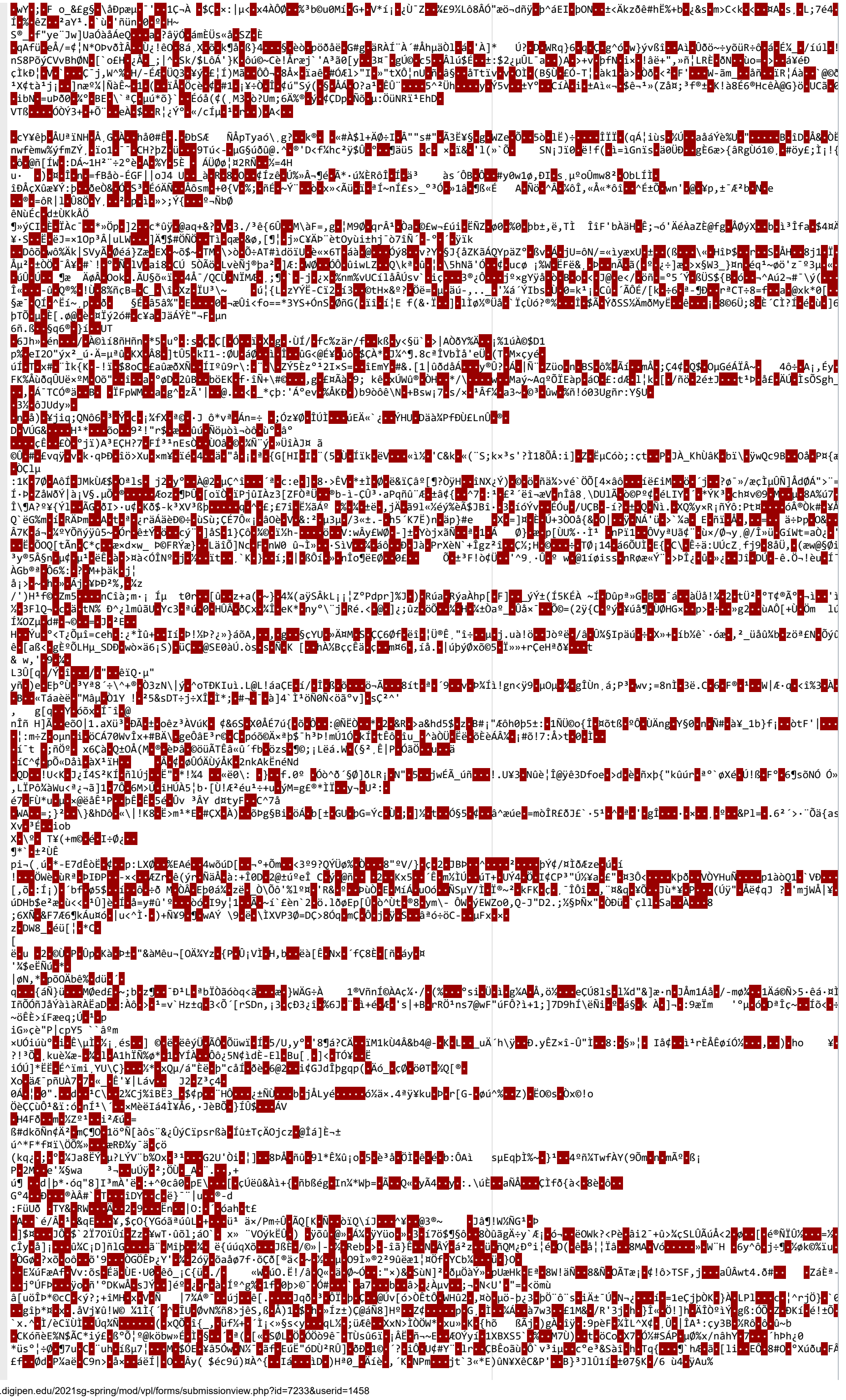
```
\return The free list.

*/
/*************************************************************************/
const void* ObjectAllocator::GetFreeList() const
{
    return FreeList_;
}

/*************************************************************************/
/*!
\fn     const void* ObjectAllocator::GetPageList() const

\brief  Returns the page list, a linked list that chains all the page.

\return The page list.

*/
/*************************************************************************/
const void* ObjectAllocator::GetPageList() const
{
    return PageList_;
}

/*************************************************************************/
/*!
\fn     OAConfig ObjectAllocator::GetConfig() const

\brief  Returns a copy of the current configuration.

\return The copy of the current configuration

*/
/*************************************************************************/
OAConfig ObjectAllocator::GetConfig() const
{
    return OAConfig_;
}

/*************************************************************************/
/*!
\fn     OAStats ObjectAllocator::GetStats() const

\brief  Returns a copy of the current stats.

\return The copy of the current stats

*/
/*************************************************************************/
OAStats ObjectAllocator::GetStats() const
{
    return OAStats_;
}

/*************************************************************************/
/*!
\fn     void ObjectAllocator::addPage()

\brief  Allocate new page. Checks if it has hit the maximum page allocated
        allowed to allocate.
*/
/*************************************************************************/
void ObjectAllocator::addPage()
{
    //Check if maximum pages is reached
    if(OAConfig_.MaxPages_ && (OAStats_.PagesInUse_ == OAConfig_.MaxPages_))
    {
        throw OAException(OAException::E_NO_PAGES, "maximum page allocated");
    }

    GenericObject* newPage;

    try
    {
        //allocate memory for new page
        newPage =
        reinterpret_cast<GenericObject*>(new u_char[OAStats_.PageSize_]);

        ++OAStats_.PagesInUse_;
    }
    catch(std::bad_alloc &)
    {
        throw OAException(OAException::E_NO_PAGES,
        "Not enough memory to add new page");
    }

    //link PageList_
    newPage->Next = PageList_;
    PageList_ = newPage;

    //Putting object onto free list
    unsigned char* temp = reinterpret_cast<unsigned char*>(newPage);
    temp += sizeof(void*);

    memset(temp, ALIGN_PATTERN, OAConfig_.LeftAlignSize_);

    temp += (offset - sizeof(void*));

    for(unsigned int i = 0; i < OAConfig_.ObjectsPerPage_; ++i)
    {
        ++OAStats_.FreeObjects_;

        if(OAConfig_.DebugOn_)
        {
            memset(temp - OAConfig_.PadBytes_ - OAConfig_.HBlockInfo_.size_, 0,
            OAConfig_.HBlockInfo_.size_);

```

```
            memset(temp, UNALLOCATED_PATTERN, OAStats_.ObjectSize_);
            memset(temp - OAConfig_.PadBytes_ ,PAD_PATTERN,OAConfig_.PadBytes_);
            memset(temp + OAStats_.ObjectSize_,PAD_PATTERN,OAConfig_.PadBytes_);
        }

        if(i != OAConfig_.ObjectsPerPage_ - 1)
        {
           memset(reinterpret_cast<unsigned char*>(temp) + OAStats_.ObjectSize_
           + OAConfig_.PadBytes_, ALIGN_PATTERN, OAConfig_.InterAlignSize_);
        }

        reinterpret_cast<GenericObject*>(temp)->Next = FreeList_;
        FreeList_ = reinterpret_cast<GenericObject*>(temp);

        temp += OAStats_.ObjectSize_ + 2* OAConfig_.PadBytes_ +
        OAConfig_.HBlockInfo_.size_ + OAConfig_.InterAlignSize_;
    }
}

/*****************************************************************************/
/*!
\fn     void ObjectAllocator::updateOAStats()

\brief  Increment stats every time an object is allocated.
*/
/*****************************************************************************/
void ObjectAllocator::updateOAStats()
{
    ++OAStats_.Allocations_;
    ++OAStats_.ObjectsInUse_;

    if(OAStats_.ObjectsInUse_ > OAStats_.MostObjects_)
        OAStats_.MostObjects_ = OAStats_.ObjectsInUse_;

    --OAStats_.FreeObjects_;
}

/*****************************************************************************/
/*!
\fn     size_t ObjectAllocator::getPageSize()

\brief  Returns the page size

\return The page size
*/
/*****************************************************************************/
size_t ObjectAllocator::getPageSize()
{
    //check if left alignment needs to be calculated
    if(OAConfig_.Alignment_ > 1)
    {
        //calculate byte needed for left alignment
        size_t left_bytes = sizeof(void*) + OAConfig_.HBlockInfo_.size_ +
        OAConfig_.PadBytes_;

        OAConfig_.LeftAlignSize_ = OAConfig_.Alignment_ -
        (static_cast<unsigned>(left_bytes) % OAConfig_.Alignment_);

        //calculate byte needed for inter alignment
        size_t inter_bytes = OAStats_.ObjectSize_ + OAConfig_.HBlockInfo_.size_
        + OAConfig_.PadBytes_ * 2;

        OAConfig_.InterAlignSize_ = OAConfig_.Alignment_ -
        (static_cast<unsigned>(inter_bytes) % OAConfig_.Alignment_);
    }

    size_t size = 0;

    //left alignment size
    //nextP + al (left alignment). Only first block has left alignment
    size = sizeof(void*) + OAConfig_.LeftAlignSize_;

    //InterAlignSize_
    size+= OAConfig_.ObjectsPerPage_* (OAConfig_.HBlockInfo_.size_ +
    OAConfig_.PadBytes_ + OAStats_.ObjectSize_ + OAConfig_.PadBytes_);

    size += (OAConfig_.ObjectsPerPage_ - 1) * OAConfig_.InterAlignSize_;

    return size;
}

/*****************************************************************************/
/*!
\fn     bool ObjectAllocator::isPageEmpty(GenericObject* page) const

\brief  Given a page, check if the page is empty by running through the
        freelist and check if there are free items in the page.

\param page - A page to check if it is empty

\return Return true is page is empty, otherwise, return false.
*/
/*****************************************************************************/
bool ObjectAllocator::isPageEmpty(GenericObject* page) const
{
    GenericObject* freeList = FreeList_;

    unsigned freeWithinPage = 0;

    while(freeList)
    {
        if(inPage(page, reinterpret_cast<u_char*>(freeList)))
        {
            ++freeWithinPage;

            if(freeWithinPage >= OAConfig_.ObjectsPerPage_)
            {
```

```
750             return true;
751         }
752     }
753     freeList = freeList->Next;
754 }
755 return false;
756 }
757
758 /*****************************************************************************/
759 /*!
760 \fn     void ObjectAllocator::freePage(GenericObject* temp)
761
762 \brief  Given a page, free memory by returning it to OS. Removes all data block
763         from the freelist, then decrement the stat counter.
764
765 \param temp - A page to free.
766 */
767 /*****************************************************************************/
768 void ObjectAllocator::freePage(GenericObject* temp)
769 {
770     removePgObjInFreeList(temp);
771     delete[] reinterpret_cast<u_char*>(temp);
772     --OAStats_.PagesInUse_;
773 }
774
775 void ObjectAllocator::removePgObjInFreeList(GenericObject* pageAddress)
776 {
777     GenericObject* temp = FreeList_;
778     GenericObject* previous = nullptr;
779
780     //If head node itself hold key or multiple key
781     while(temp!= nullptr && inPage(pageAddress,reinterpret_cast<u_char*>(temp)))
782     {
783         //Change head
784         FreeList_ = temp->Next;
785
786         //Change temp
787         temp = FreeList_;
788         --OAStats_.FreeObjects_;
789     }
790
791     //Delete occurence other than head
792     while(temp != NULL)
793     {
794         //Search key to be deleted
795         //Keep track of previous node, need to change previous->Next
796         while(temp != nullptr &&
797         !inPage(pageAddress, reinterpret_cast<u_char*>(temp)))
798         {
799             previous = temp;
800             temp = temp->Next;
801         }
802
803         //If key not present in linked list
804         if(temp == nullptr) return;
805
806         //unlink node from linked list
807         previous->Next = temp->Next;
808
809         --OAStats_.FreeObjects_;
810
811         //Update temp for next loop
812         temp = previous->Next;
813     }
814 }
815
816 /*****************************************************************************/
817 /*!
818 \fn     bool ObjectAllocator::inPage(GenericObject* pageAddress,
819         u_char* address) const
820
821 \brief  Given a page address and any address, check if address is in the page.
822
823 \param  pageAddress - The pointer to the page.
824 \param  address - The address to check if it is in range.
825
826 \return Returns true if in page, otherwise, return false.
```

index.chm

[illegible]



VPL

◄ General Information

Jump to...

Introduction to Data Structures ►